DOML - a Compiler Environment for Dynamic Optimization Supporting Multiple Solvers*

Tomasz Tarnawski   Radosław Pytlak
Warsaw Technical University, Institute of Automatic Control and Robotics
ul. św Andrzeja Boboli 8, 02-525 Warsaw

Abstract

The Modelica language may serve well as a base for defining optimal control problems, given a few relatively minor syntax extensions. One example proving that point is Optimica and another one is DOML (Dynamic Optimization Modeling Language) – installed on IDOS (Interactive Dynamic Optimization Language) and described in this paper. The DOML implementation is, actually, heavily based on the (open source) compiler of Optimica but it provides a number of important features absent in its precursor. One main extension of the compiler lies in its built-in mechanism supporting the use of many different optimization solvers (selected on the fly, depending on the content of the problem definition) and to seamlessly add new, external, solvers. In result, the range of problems that may be specified with DOML and solved in the IDOS environment is quite wide and keeps growing. The scope of problems ranges from some static optimization problems through regular ODE, parametric optimization, minimum time problems, up to DAE with higher index. DOML language extensions also provide preliminary support for multi-objective optimization and PDE problems.

Keywords: dynamic optimization; optimal control; Optimica

1 Introduction

Dynamic Optimization Modeling Language (DOML) was initially proposed ([23]) as a programming-language-independent communication format for newly developed Interactive Dynamic Optimization Server (IDOS, described e.g. in [22]). The IDOS server, envisioned by its proponents as "the NEOS for optimal control", provides an on-line environment for dynamic optimization. It receives inputs — definitions of dynamic optimization problems specified in DOML — from its users and responds with solutions (if) found. Although still under development, IDOS is already operational, on-line at [32]. The main mode of working with the server is through a web browser, where the (logged-in) user may type problem definitions in DOML, submit tasks, view results etc. The elements and functions of the IDOS server, including exemplary computational results, have already been described in a few publications (e.g. in [22], [23] and [24]) and, in particular, in a parallel paper to be presented on the very same 10^{th} Modelica Conference ([25]). Hence, here the functioning of IDOS itself will not be discussed any further, the core focus goes to the DOML format and its implemented compiler (as installed on IDOS).

The DOML format provides a mean for defining optimal control problems in a slightly extended Modelica syntax. In that respect it is very similar to Optimica (initially proposed in [2]) and in fact implementation of its compiler[1] is based on the (open source) Optimica compiler, available at [31]. Since Optimica itself has been presented in numerous papers, in particular on the Modelica Condeference (e.g. [3]), it is not discussed here in much detail. Instead, the focus lies on the main features of DOML that set it apart from Optimica.

As stated above, DOML compiler is installed on a on-line–accessible server ([32]) and is used to interpret the input form (remote) users entering problems they want solved. In contrast, Optimica compiler comes as a part of a downloadable and locally installed JModelica.org environment, where the user is mainly exposed to Python

*The work presented in the paper has been partially supported by NCBiR grants: R02-0009-06, PBS1/A7/6/2012

[1]DOML compiler is developed as open source. Contact the authors for setting up access to code repository

programming console as a mode of working with compiled problems, numerical packages and optimization solvers (see e.g. [4]). That architectural contrast dictates the main difference between the two compilers: DOML compiler provides inherent, built-in support for generating code for a multitude of different solvers and numerical packages (and new solvers may still be added to the environment) while in JModelica.org environment implementing any extensions or different optimization algorithms is (only) possible though Python scripts (see [26] for an example) – executed *on top of* the Optimica compiler.

One design principle for the IDOS server was its ease of use, particularly for users with limited programming skills. Hence, it seemed highly desirable to avoid any meanders on the road form a problem to its solution, such as the need of using other (programming) languages, learning environment's internal API or studying interfaces to numerical libraries. In effect, all that an IDOS user must provide is the problem definition in DOML and DOML is then the only language that one really needs to comprehend in order to use the server. (However, as discussed later, an advanced user with good acquaintance with the compiler's and server's internals has also the ability to extend the server's functionality and to implement own solving algorithms).

We find the need for implementing support for multiple solvers to be justified and intuitive. For once, dynamic optimization problems come in many different kinds (e.g. parametric, minimum time, DAE with higher index, etc) and so one solver would not be general enough to solve all of them efficiently and accurately; solvers dedicated to particular problem kinds simply do the job better. The second reason is to open the possibility for applying "solver chaining" i.e. a strategy of using two (or more) solvers consecutively on one problem. The first one yields a crude approximation while the next guarantees a more accurate solution but requires a reasonably good initial guess (and possibly other warm-starting information) – that may be taken form the first solver. Automation of such procedure seems very tempting, yet it poses a number of challanges and in same cases may plainly be impossible (e.g. due to the nature of information needed by some second-line solvers, such as [17]). Currently, a user must recode the problem manually through editing in the results obtained form the first solver and resubmit the file. For more details on chaining solvers in IDOS see e.g. [24].

The paper is organized as follows. The following section 2 provides a general introduction to the DOML compiler. Section 3 proceeds towards the technical aspects dealing with the paper's key issue — implementation of multi-solver support. Additional attention is devoted to the more advanced topic of supplementing the compiler with external code generators (section 3.1). Next, in section 4, the set of currently deployed solvers is briefly presented, together with (selected) code generators within the compiler implemented to couple with these solvers. Finally, section 5 provides supplementary information on other important features proposed in DOML (and setting it apart from Optimica) yet less tightly related to multi-solver support.

2 Compiler Overview

On the most general level, the working of DOML compiler is identical to that of Optimica. It reads in an input `.mo` file into memory, where it creates own representation of all elements of the problem's definition. Based on that, it generates output file(s) written in a regular programming language (mostly C/C++) — therefore the compiler, formally, is actually more of a translator (from Modelica to C/C++). The produced files may then be placed as input to the standard (`gcc`) compiler and linker (bringing in required external libraries) to produce an executable file that will carry out the actual solving algorithm.

The generation of C (C++) code is template-driven: in order to produce any output, the compiler reads in an external *template* file – resembling closely a regular C++ file where only certain spots are specially marked (with so called *tags*). These tags represent places in the code where external snippets of code are to be pasted, e.g. the `$n_real_x$` tag will be replaced with the number of state variables defined in the model. The code generating method reads in the template file(s) and reproduces it to the output. Every time it comes across a tag, however, it replaces it in the output with an appropriately elaborated fragment of code — depending on the tag it may be a single number literal or several lines of C/C++ code. A special mechanism dispatches the calls to the right code (or: snippet) generating methods based on the textual content of the tag.

The compiler is fully implemented in Java. Its executable, *together with all required template files*, is packed into a single Java `jar` file: `DOMLCompiler.jar` — which is where the differences against Optimica start. Typical execution of the compiler, on an exemplary input file `theInput.mo` , would take the form of:

```
> java -jar DOMLCompiler.jar theInput.mo
-o outDir
```

and all produced output would end up in `outDir` directory. It proved very convenient to supplement the set of templates with a `Makefile` so that in the output folder, next to C/C++ source code (and possibly other auxiliary) files, a `Makefile` would also be present. Then, simply running

```
> make
```

in that directory produces the executable to run and solve the problem.

The set of generated files constitutes, essentially, a problem-specific implementation of a dynamic optimization algorithm and most usually it makes an extensive use of (external) numerical libraries — through their APIs. Naturally, the generated files (and, in particular, the build commands in `Makefile`) must closely correspond with the libraries installed on the computer. As the compiler provides the problem-specific information, its back-end — the code generating *engine* — becomes responsible for "mating" the compiler with the solver(s). The two *pistons* of that engine are: the collection of template files (mentioned above) and the Java class implementing the logic for producing code snippets for each tag. Then their common *crankshaft*, to keep with the comparison, would be the set of tags present in the templates and rendered by the Java class (technically, it is in fact a fair number of small internal classes defined within the main code generator class; each of them is responsible for interpreting its separate tag).

The design of the DOML compiler provides for an easy way for adding and/or replacing backends. Such back-end block, composed of a code generating class together with its related templates, can therefore be viewed as a plug-in to the compiler (especially, since it can be packaged into a completely separate `jar` file, as described in detail in section 3.1). The concrete code generator is selected and loaded on the fly, based on the content of the input `.mo` file. Irrespective of the code generator eventually employed, the mode of executing problems under DOML is always the same: first run `jar`, then `make`.

In that respect, DOML is very different form Optimica which contains only built-in (and "hardcoded") code generators – to C and to XML. The C generator targets a single, specific optimal control solver environment (contained within JModelica.org environment). Admittelly, the XML code generator grants the user more flexibility, as the produced files may be imported by other tools (as discussed in [1]). The flipside of that approach is that it burdens the user with handling yet another file format (XML, this time) that he or she has to transport from one environment to another – either manually or through Python scripting (not mentioning, that such additional tools would then also need to be installed). This may be perfectly fine for users proficient with programming, scripting and managing multiple tools installed on their computers, but not all of them are. As already indicated, IDOS targets different users – ones, that would prefer not to (learn and) do all that, and instead to have one simple tool (a web browser) and one language (DOML) for solving all their (optimal control) problems.

3 Implementation of Multiple-solver support

In the context of DOML compiler, the term *solver* refers to a set of templates together with numerical libraries used by them and needed to build the executable. In other words, each solver is an implementation of a particular dynamic optimization algorithm; as soon as it is filled with problem specific data it may be compiled and executed. The actual collection of solver packages deployed on IDOS server (and required libraries installed alongside) is discussed in [25] and here touched upon in section 4. For the purpose of the current discussion, it is sufficient to recognize that there may be present quite a few different packages implementing algorithms for solving distinct kinds of optimization problems and with time still new packages may be implemented and plugged-in to the environment.

In DOML compiler environment each solver's code generating class — i.e. the class containing the logic to fill up all templates for particular

```
package pl.pw.DOML.codegen;

public abstract class BaseDOMLCGen
        extends OptimicaCGenerator {

  public static BaseDOMLCGen
    loadGenerator(String solverName,
                  FDOMLClass fc,
                  Properties props) {
     ...
  }

  public abstract ArrayList<String>
    getTemplateFNames();

  public void
    generate(String templateF,
             String outF) { ... }

   ...

}
```

Listing 1: BaseDOMLCGen class excerpt

solver — must be a subclass of `BaseDOMLCGen`. Excerpts of this class, shown on listing 1, present its most important features. It:

- implements static method `loadGenerator()` for searching, loading and instantiating the appropriate, concrete code generating class. It utilizes Java reflection mechanism to look for proper compiled Java classes.

- declares `getTemplateFNames()` abstract method thet every generator must override to communicate (through call-back) its set of template file names to the compiler. This way, each code generator may be tied to its own, completely separate set of templates.

- implements `generate()` method that carries out the process for each template file (whose name is provided as the first parameter) by reading it and dispatching calls to individual tag-generating methods.

Any code generator class derived from `BaseDOMLCGen` needs to implement only the `getTemplateFNames()` method and the logic of generating code snippets for its specific tags (i.e.: tags present in templates associated specifically with that generator).

The class searching and loading mechanism relies on the use of Java annotation `@Solver` defined specifically for that purpose. Each code generator implemented according to the description above must be appropriately annotated, where at least the solver's unique name must be given — as on the example below with `"olado"` :

```
@Solver("olado")
public class DOMLCGen4Olado
        extends BaseDOMLCGen { ...
```

All template files used by that solver must then reside in the `jar` file, inside its `/templates/olado` subdirectory.

The annotation `@Solver` also provides means for specifying the kinds of problems that a given solver package can handle, through a set of attributes defining whether the package requires (`TRUE`), accepts (`DONT_CARE`) or disallows (`FALSE`) certain qualities and elements of a problem. The default value for all `@Solver`'s attributes is `FALSE`, hence the ones not applicable may be omitted. Example below illustrates the point:

```
@Solver(value = "rkcon",
   freeFinalTime
           = ApplicableTo.TRUE,
   minFinalTime
           = ApplicableTo.DONT_CARE,
   residualEquations
           = ApplicableTo.DONT_CARE,
   finalStateConstraints
           = ApplicableTo.DONT_CARE)
public class DOMLCGen4RKcon
        extends BaseDOMLCGen {
```

As of now, this annotation's attributes describe, whether the given solver is applicable to:

- with respect to the problem's structure — static optimization, ODE or DAE systems, parametric optimization (in particular: optimization of the system's initial state), PDE problems;

- with respect to the elements present in the definition — problems with: free final time (in particular, where final time is the objective), equations specified in residual form, integer-valued decision variables, multiple objectives;

- with respect to the kinds of constraints defined — box constraints, constraints on the final state, general functional constraints or non-stationary constraints.

This setup should not yet be considered final but it is well past the proof-of-concept stage. Having the above list as a rough guidance, the goal of development efforts of the IDOS server is to furnish a host of solvers covering typical cases of dynamic optimization problems. The current state of efforts, as far as code generators implemented and the set of corresponding libraries installed, is briefly discussed in section 4.

When executing a problem specified in DOML, one may force the use of a particular package through (Modelica-like) annotation `solver` handled by the DOML compiler, as so:

```
annotation(solver="olado");
```

If such annotation is omitted in the DOML input (or solver is specified as `"auto"`), the compiler makes its best effort to load the generator class that fits the current problem. Upon entering the code generation stage (i.e. after parsing input file, doing semantic analysis and problem transformations e.g flattening or alias elimination) the compiler queries the object containing the AST representation of the input — to classify the problem along the lines sketched above. Then, it browses available code generator classes (subclasses of `BaseDOMLCGen`) and tries to match their `@Solver` description to that classification. If an exact counterpart cannot be found, a simple (preliminary) heuristic is used to choose the closest match. An object of the selected class is instantiated and used thereupon.

3.1 External code generators

In the mechanism described above, the logic responsible for browsing for available code generator classes implements one additional, powerful feature. The searching is not limited to the content of the `DOMLCompiler.jar` file, but in fact incorporates all `jar` files found in the current folder. This way any external solver, packaged (together with its templates) into a `jar` file, may equally well be used by the compiler task. An advanced user may therefore implement own solver `jar` package (which is the harder part) and add it seamlessly and non-intrusively to the compiler (which then is as easy as uploading the `jar` file on the server).

When implementing own solver package, say under the name `mySolverPack`, one must produce a Java `jar` file that fulfills the following minimum requirements:

- it must contain the code generator class — subclass of `BaseDOMLCGen` — annotated as `@Solver("mySolverPack")` and implementing `getTemplateNames()` that returns a collection of names of solver's template files (including a `makefile`);
- it must contain all template files placed in `/templates/mySolverPack` subfolder;
- templates may use only numeric libraries installed on the server (installing extra libraries on user accounts is not yet supported), the `makefile` should be written accordingly;
- the code generating class must handle appropriately solver-specific tags appearing in the templates.

Preferably, the annotation `@Solver` should also specify applicability of that package (as discussed earlier); otherwise, one will be able to use the solver only by specifically requesting its name through annotation in the DOML input file.

The above requirements clearly imply, that this functionality is meant for advanced users–developers of optimization algorithms. Preparing the set of templates calls for a good knowledge (and at least some practice of use) of libraries installed on the server while implementing own code generator class is possible only with good familiarity of the compiler's internal API (in particular: the data structure representing the compiled input problem).

4 Implemented solvers and code generators

At the moment the `IDOS` server can handle control problems described with ordinary equations and differential–algebraic equations but the incorporation into the `IDOS` solvers for problems with partial differential equations is also under way.

The IDOS server enables solving optimal control problems by using essentially different methods ranging from ones based on *a'priori* discretization through utilizing variable stepsize integration and adjoint equations to shooting methods applied to differential equations derived from necessary optimality conditions. These methods and solvers are discussed in some detail in [25].

Obviously, not all solvers can be used on every problem. The compiler is fit to cooperate with the installed solvers, through the following code generator classes implemented in DOML compiler:

- `DOMLCGen4Olado` — ODEs are discretized *a'priori* and then the large–scale NLP problem solved by a static optimization solver (`Ipopt` [29], `KNITRO`, etc)–the library `OLADO` is described in [5] and uses several interior point methods ([16],[11]) for QP problems to which solvers refer;
- `DOMLCGen4cvodes` — ODEs treated as continuous time (adaptive stepsize integration using `SUNDIALS`, [15]), the optimization problem solved with the help of adjoint equations (in the reduced space) and the `Ipopt` package–the solver is based on `cvodes` procedure ([28]) from `SUNDIALS`;
- `DOMLCGen4bndsco` — indirect shooting method using the `BNDSCO` package developed by Oberle ([17]) ;
- `DOMLCGen4Pantelides` — higher index DAEs, implementing the Pantelides' graph based initialization algorithm ([19]), also `Maxima` package for symbolic differentation is used ([30]);
- `DOMLCGen4OladoBonmin` — ODEs with integer valued controls, solved through *a'priori* discretization of ODEs as applied in the `OLADO` package and with the help of MIP solvers from `BONMIN` package ([6],[7]);
- `DOMLCGen4cvodesBonmin` — ODEs with integer valued controls, the optimization problem solved with the help of adjoint equations (evaluated by `SUNDIALS` programs) and `BONMIN` package;
- `DOMLCGen4OladoHqpOmuses` — various dynamic optimization solvers based on `HQP` and `OMUSES` packages ([9],[10]);
- `DOMLCGen4RKon` — index one DAEs treated by implicit Runge–Kutta scheme, optimization problem solved in reduced space by `RKCON` solver (special treatment of state constraints)–the justification for the solver `RKCON` is presented in [20];
- `DOMLCGen4Radau` — up to index three DAEs treated by `RADAU5` (implicit R-K) solver ([14],[12],[13]), optimization problem solved in reduced space by `RKCON` solver, the package is based on adjoint equations for higher index DAEs derived in [21].

The above list is not exhaustive as, for instance, a few of other back-ends undergo testing and/or are under construction — e.g. `DOMLCGen4PDE` or `DOMLCGen4staticIpopt` (where names imply their purpose).

5 Other enhancements in DOML

The main direction of extending DOML syntax revolved around the idea of "chaining" of solvers, i.e. using a number of solvers in a sequence so that a succeeding solver would (use as an initial guess and) improve on the solution arrived at by its predecessor. Usually an approximate solver would be used first, followed by one that is more accurate yet also more sensitive (susceptible) to the choice of starting point.

One of DOML language proposed features is an external package added into the Modelica Standard Library — `Modelica.DOML` — with its predefined (/built-in) data types and functions. Currently `Modelica.DOML` contains two subpackages: `internal` and `Inputs`. Importantly, the definitions in `Modelica.DOML.internal` are automatically visible in any DOML file i.e. it is always implicitly imported by the compiler (therefore, specifying explicitly **`import`** `Modelica.DOML.internal.*;` would be redundant and will have no additional effect). The package `internal` contains, among others, the definition of **`Formula`** class needed for certain aspects of solver chaining (but also e.g. **`Domain`** class used in defining PDE problems). In turn, the purpose of `DOML.Inputs` was to allow for easy definition of known (predefined) signals, most notably with `Spline` – using tabular values to produce piece-wise constant ($\texttt{degree} = 0$) or continuous ($\texttt{degree} \geq 1$) signal, based on polynomial interpolation.

A simple and intuitive language extension (or perhaps just a bug fix to Optimica) was to allow, in certain cases, *continuous* variability in the right-hand-side expression defining the value of the `initialGuess` attribute. If a variable's value can change over time, then it seams reasonable to allow the user to provide an initial guess for it that also is not constant over time. After that enhance-

```
Modelica.DOML.Inputs.Spline
  u_init(startTime=0, finalTime=1,
         table=[0,0; 0.2,.096;
         0.4,.1; 0.6,.1;
         0.8,.096; 1,0],
         degree = 1);
Real u(initialGuess = u_init.y);
```

Listing 2: Specifying on initial guess signal for a continuous variable

```
class Formula
  parameter Boolean linear;
  Real lagrange(dual = true);
  parameter Integer order;
end Formula;
```

Listing 3: Definition of the predefined class **Formula**.

ment, the above-mentioned Spline may be used to feed an arbitrary signal as an initial guess of a variable, as illustrated on listing 2. This way, a user can run optimization with a fairly good starting point, for instance obtained by pre-solving the problem with a different (simpler) solver or with coarser discretization — a typical implementation of the strategy of "chaining of solvers".

Next to the (near optimal) trajectories found for variables and controls, other (by)products of (pre)solving an optimal control problem may be available and beneficial when warm-starting a high-precision solver. Among such, are the values of adjoint variables and Lagrange multipliers arrived at near the solution. (Adjoint variables and Lagrange multipliers are, arguably, two names representing the same fundamental concept hence, for the sake of simplicity, for both cases the syntax provides the same name: lagrange). Assigning the lagrange values to equations (/constraints) needs a mechanism for identifying particular formulas. In DOML it is devised by means of labeling them, as so

```
eq_x1: der(x1) = p1*x1 - p2*x2*x3;
```

with simultaneously declaring the "label variable" as **Formula** eq_x1;. The compiler checks if all variables of type **Formula** are referenced as labels and, conversely, that all labels used next to equations are appropriately declared variables. The **Formula** class is declared in DOML.internal, as shown on listing 3.

The objects of the new predefined type **Formula** are meant to be used to specify attributes describing the particular equation or constraint – most notably the above mentioned adjoint variable/Lagrange multiplier, but for prospective use other attributes were defined as well. Similarly to the earlier example, one can now specify an initialGuess for the lagrange attribute associated with a particular equation or constraint. Such additional information may be benefitial, or plainly required, in certain solvers (e.g. BNDSCO).

The last to discuss modification of DOML vs. Optimica lies in a slightly different way of defining the objective within an **optimization** class. DOML breaks up with Optimica's objective attribute and introduces two keywords, **minimize** and **maximize**, in its stead. Each has an almost identical meaning to Modelica's **parameter** keyword, but in addition signals that the given parameter holds the objective value.

The modification was motivated by its following advantages:

- it clearly indicates the direction of optimization; in some domains the default of minimization may not be as natural.
- objective variables can now have meaningful names;
- it becomes possible (and strikingly simple) to define multi-objectives problems in a natural way e.g. with:
  ```
  maximize Real profit =... ;
  minimize Real deliveryTime =... ;
  ```
  (implementing a multi-objective solver is, naturally, quite another issue)
- it makes it possible to define problems with integer valued (or even Boolean valued, for that matter) objective functions (again, we are not touching on the implementation aspect here).

A number of other important language extensions is related to using DOML to specify and (prospectively) solve PDE problems. This aspect also deserves attention but is clearly beyond the scope (and page allowance) of this paper. This aspect of implementation is described in [23] and was partially inspired by [27].

6 Summary and future work

IDOS (Interactive Dynamic Optimization Server) provides an environment for defining and solving optimal control problems in DOML (Dynamic Optimization Modeling Language) — a modeling language derived from Modelica (/Optimica). The main focus of the paper was put on these features of the server's environment — in particular, of the DOML compiler — that provide support for applying different solvers to different problems. Additional information about the current range of implemented solvers was also provided.

A number of features proposed in DOML set it apart from Optimica. The authors believe, that these modifications deserve to be called "improvements" and/or "extensions" as they seem to enhance both expresiveness and applicability of the language. Their presentation and discussion is scattered throughout the paper and therefore, for clarity and readibility, here we (re)state them all together in a concise manner. The new features of DOML are:

- implementation of multiple solvers with a mechanism for adding (plugging-in) new code generators (vs. hard-coded, closed set of code generators in Optimica);
- **`annotation`**`(solver )` allowing the user to choose a particular solver package to be used;
- support for continuous variablity of the `initialGuess` attribute (vs. parameter variability in Optimica) adds flexibility in defining initial (warm-starting) point of computations, e.g. with using provided `Spline` signal generator;
- mechanism for labaling equations and constraints (missing in Modelica/Optimica);
- mechanism for referencing (and warm-starting) adjoint variables / Lagrange multipliers of (labelled) formulas;
- keywords **`maximize`** and **`minimize`** defining the objective function with indicating the direction of optimization (vs. `objective` attribute in Optimica)
  note: makes it possible to define multiple-objective problems, now the objective does not have to be real-valued.

Apart form the above list, the lion's share of the DOML compiler functionality is inherited directly form the (JModelica.org) Optimica compiler whose authors, naturally, deserve due recognition and honor.

Currently, significant parts of the server's functionality are still under development and testing. There is still much room for improvement, for instance as far as robustness (e.g. in cases of some less usual formulations that are acceptable from the language point of view but not recognized by code generating rules) or graceful error handling are concerned. A further reaching to-do list also contains: polishing the mechanism for problem classification and choosing solvers/generators (more advanced rules for matching most appropriate solver to a given problem) and better support for chaining of solvers (possibly, through producing solver's output in the form of "enriched" input, with solution pasted in as, e.g. the value of control's `initialGuess` attribute; such output could then be directly used to warm-start another solver).

One central goal in the design of the environment was to provide mechanisms for easily extending the compiler's code generating functionality in order to support a growing number of solver packages. As discussed in some detail above, the goal has already been reached successfully. In result, the compiler offers a remarkable feature: extensions of its code generating functionality is possible virtually on the fly — by simply adding an external `jar` file and without any interference with the compiler's executable. With this mechanism, remote users of IDOS may be enabled to implement own solver packages and deploy them on the server. If the proposed solution catches on and stands the test of time, IDOS could become an important, extensible, multi-user environment for solving wider and wider range of optimal control problems — "the NEOS of optimal control".

References

[1] Andersson J, Åkesson J, Casella F, Diehl M. Integration od CasADi and JModelica.org. In: Proceedings of the 8th Modelica Conference 2011, Dresden, Germany, Modelica Association, 20-22 March 2011.

[2] Åkesson J. Tools and Languages for Optimization of Large-Scale Systems. Lund, Swe-

den: *PhD thesis*, Department of Automatic Control, Lund University, 2007.

[3] Åkesson J. Optimica–an extension of Modelica supporting dynamic optimization. In: Proceedings of the 6th Modelica Conference 2008, Bielefeld, Germany, Modelica Association, 3-4 March 2008.

[4] Åkesson J, Gäfvert M, Tummescheit H. JModelica—an Open Source Platform for Optimization of Modelica Models. In: Proceedings of MATHMOD 2009 - 6th Vienna International Conference on Mathematical Modelling, Vienna, Austria, TU Wien, February 2009.

[5] Błaszczyk J, Karbowski A, K. Malinowski K. Object Object library of algorithms for dynamic optimization problems: benchmarking SQP and nonlinear interior point methods. In: Journal of Applied Mathematics and Computer Science, Vol. 17, 515–537, 2007.

[6] Bonami P, Wachter A, Biegler L.T, Conn A.R, Cornujols G, Grossmann I.E, Laird C.D, Lee J, Lodi A, Margot F, Sawaya N. An algorithmic framework for convex mixed integer nonlinear programs. In: Discrete Optimization, Vol. 5, 186–204, 2008.

[7] Bonami P, Lee J BONMIN Users' Manual, 2009.

[8] Bonami P, Lodi A, Cornujols G, Margot F. A feasibility pump for mixed integer nonlinear programs. In: Mathematical Programming, Vol. 119, 331–352, 2009.

[9] Franke R. Anwendung von Interior-Point-Methoden zur Lösung zeitdiskreter Optimalsteuerungsprobleme. *Master's thesis*, Techniche Universität Ilmenau, Fakultät für Informatik und Automatsierung, Institut für Automatisierungs- und Systemtechnik Fachgebiet Dynamik und Simulation ökologischer Systeme, Ilmenau, Germany, October 1994 (in German).

[10] Franke R. OMUSES — a Tool for the Optimization of MUltistage Systems and HQP — a Solver for Sparse Nonlinear Optimization. Dept. of Automation and Systems Engineering, Technical University of Ilmenau, Germany, September 1998.

[11] Gondzio J. Multiple centrality corrections in a primal-dual method for linear programming. *Technical Report 1994.20*, Department of Management Studies, University of Geneva, Geneva, Switzerland, November 1994.

[12] Hairer E, Lubich Ch, Roche M. The numerical solution of differential–algebraic equations by Runge–Kutta methods. In: Lecture Notes in Mathematics 1409, Springer–Verlag, Berlin, Heidelberg, 1989.

[13] Hairer E, Norsett S.P, Wanner G. Solving Ordinary Differential Equations I., Springer–Verlag, Berlin Heidelberg New York, 2008.

[14] Hairer E, Wanner G. Solving Ordinary Differential Equations II, Springer–Verlag, Berlin Heidelberg New York, 1996.

[15] Hindmarsh, A.C., Brown, P.N., Grant, K.E., Lee, S.L., Serban, R., Shumaker, D.E., Woodward, C.S. SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers, preprint, UCRL-JRNL-200037, 2004.

[16] Mehrotra S. On the implementation of a primal–dual interior point method. In: SIAM Journal on Optimization, Vol. 2, 575–601, 1992.

[17] Oberle H.J, Grimm W. BNDSCO – A Program for the Numerical Solution of Optimal Control Problems, Report No. 515 der DFVLR (German Test and Research Institute for Aviation and Space Flight), 1989.

[18] Oberle H.J. Numerical Solution of Minimax Optimal Control Problems by Multiple Shooting Technique. In: J. Optimization Theory and Applications, Vol. 50, 331–364, 1986.

[19] Pantelides C. Consistent initialization of differential–algebraic system. In: SIAM J. Scientific and Statistical Computation Vol. 9, 213-231, 1988.

[20] Pytlak R. Numerical procedures for optimal control problems with state constraints. Lecture Notes in Mathematics 1707, Springer–Verlag, Heidelberg, 1999.

[21] Pytlak R. Numerical procedure for optimal control of higher index DAEs. In: J. Discrete and Continuous Dynamical Systems, Vol. 29, 1–24, 2011.

[22] Pytlak R, Tarnawski T, Fajdek B, Stachura M. Interactive dynamic optimization server–connecting one modeling language with many solvers. In: Optimization Methods and Software, 2013. `http://dx.doi.org/10.1080/10556788.2013.799159`.

[23] Pytlak R (ed.). Interactive computer environment for solving optimal control problems–IDOS. *final Research Report, under the grant R02-0009-06*, Warsaw, Poland, 2012.

[24] Pytlak R, Błaszczyk J, Krawczyk K, Tarnawski T. Solvers chaining in the IDOS server for dynamic optimization. In: Proceedings of 52nd Conference on Decision and Control, Florence, Italy, 10-13 December 2013.

[25] Pytlak R, Tarnawski T. IDOS – (also) a Web Based Tool for calibrating Modelica model. Accepted for: 10th Modelica Conference 2014, Lund, Sweden, Modelica Association, 10-12 March 2014.

[26] Rantil J, Åkesson J, Führer C, Gäfvert M. Multiple-Shooting Optimization using the JModelica.org Platform. In: Proceedings of the 7th Modelica Conference 2009, Como, Italy, Modelica Association, 20-22 September 2009.

[27] Saldamli L. PDEModelica – A High-Level Language for Modeling with Partial Differential Equations, *PhD. thesis*, Department of Computer and Information Science, Linköping University, 2006.

[28] Serban R, Hindmarsh A.C. CVODES, the sensitivity-enabled ODE solver in SUNDIALS. In: Proceedings of the 5th International Conference on Multibody Systems, Nonlinear Dynamics and Control, ASME, Long Beach, CA, 2005.

[29] Wachter A, Biegler L.T. On the implementation of an interior point Line search filter algorithm for large scale nonlinear programming. In: Math. Programming, Vol. 106, 25–57, 2006.

[30] Maxima Manual Ver 5.30.0, `http://maxima.sourceforge.net/docs/manual/en/maxima.pdf`, November 2013.

[31] Open Source Project: JModelica.org. `http://www.jmodelica.org/`. November 2013.

[32] Interactive Dynamic Optimization Server. `http://idos.mchtr.pw.edu.pl/`. November 2013.